| REPORT DOCUMENTATION PAGE | Form Approved OMB No. 0704-0188 |
|---|---|

| | | |
|---|---|---|
| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE 3/13/95 | 3. REPORT TYPE AND DATES COVERED Final Report |

| | |
|---|---|
| 4. TITLE AND SUBTITLE<br>Automated Support for Software Engineering | 5. FUNDING NUMBERS<br>$339,826.00<br>2304/FS<br>61102F |
| 6. AUTHOR(S)<br>Douglas R. Smith | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)<br>Kestrel Institute<br>3260 Hillview Avenue<br>Palo Alto, CA. 94304 | 8. PERFORMING ORGANIZATION REPORT NUMBER<br>AFOSR-TR- 95 - 0269 |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)<br>Major David Luginbuhl<br>ASOFR/NM<br>Directorate of Mathematical & Computer Science<br>110 Duncan Avenue, Suite B115<br>Bolling AFB, D.C. 20332-0001 | 10. SPONSORING/MONITORING AGENCY REPORT NUMBER<br>F49620-91-C-0073 |
| 11. SUPPLEMENTARY NOTES | DTIC ELECTE JUN 1 9 1995 S D |
| 12a. DISTRIBUTION/AVAILABILITY STATEMENT<br>Unlimited<br>This document has been approved for public release and sale; its distribution is unlimited. | 12b. DISTRIBUTION CODE<br>F |

13. ABSTRACT (Maximum 200 words)

The key idea of this project is that parameterized theories and various operations on them provide a uniform conceptual foundation for software engineering. The broad objectives of this project are (1) to show how well-understood concepts of mathematical logic apply to software engineering, (2) to identify and automate the basic operations on theories that underlie and support formal approaches to software system development and evolution, and (3) to demonstrate a higher and more broadly applicable level of software automation than has previously been achieved.

19950615 058

DTIC QUALITY INSPECTED 5

| | |
|---|---|
| 14. SUBJECT TERMS<br>Parametierized theories, software engineering, mathematical logic, formal approach to software system development | 15. NUMBER OF PAGES<br>40 |
| | 16. PRICE CODE |

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT<br>Unclassfied | 18. SECURITY CLASSIFICATION OF THIS PAGE<br>Unclassified | 19. SECURITY CLASSIFICATION OF ABSTRACT<br>Unclassified | 20. LIMITATION OF ABSTRACT<br>Unlimited |

## GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements.***

**Block 1.** Agency Use Only *(Leave blank)*.

**Block 2.** Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

**Block 3.** Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

**Block 4.** Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

**Block 5.** Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

| | | | |
|---|---|---|---|
| C | - Contract | PR | - Project |
| G | - Grant | TA | - Task |
| PE | - Program Element | WU | - Work Unit Accession No. |

**Block 6.** Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

**Block 7.** Performing Organization Name(s) and Address(es). Self-explanatory.

**Block 8.** Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

**Block 9.** Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

**Block 10.** Sponsoring/Monitoring Agency Report Number. *(If known)*

**Block 11.** Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

**Block 12a.** Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

| | |
|---|---|
| DOD | - See DoDD 5230.24, "Distribution Statements on Technical Documents." |
| DOE | - See authorities. |
| NASA | - See Handbook NHB 2200.2. |
| NTIS | - Leave blank. |

**Block 12b.** Distribution Code.

| | |
|---|---|
| DOD | - Leave blank. |
| DOE | - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports. |
| NASA | - Leave blank. |
| NTIS | - Leave blank. |

**Block 13.** Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

**Block 14.** Subject Terms. Keywords or phrases identifying major subjects in the report.

**Block 15.** Number of Pages. Enter the total number of pages.

**Block 16.** Price Code. Enter appropriate price code *(NTIS only)*.

**Blocks 17. - 19.** Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

**Block 20.** Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

# Contents

# Automated Support for Software Engineering
Final Report

P.I. Name: Douglas R. Smith
Insitution: Kestrel Institute
Telephone: (415) 493-6871
E-mail: smith@kestrel.edu
Contract Number: F49620-91-C-0073
Date: 11 March 95

## 1. Objectives

The key idea of this project is that parameterized theories and various operations on them provide a uniform conceptual foundation for software engineering. The broad objectives of this project are (1) to show how well-understood concepts of mathematical logic apply to software engineering, (2) to identify and automate the basic operations on theories that underlie and support formal approaches to software system development and evolution, and (3) to demonstrate a higher and more broadly applicable level of software automation than has previously been achieved.

The following paragraphs describe our results over the past few years. Our efforts can be loosely classified in terms of theory, experimental testbed, and applications. We use KIDS (Kestrel Interactive Development System) as our testbed and showcase for our experiments in automated software engineering. Our theoretical work both motivates new extensions to KIDS and draws on our experience with KIDS as inspiration for new topics. The interaction of these two efforts has been extremely fruitful over the past few years. The current project is based on our observation that much of what goes on in KIDS is uniformly describable in terms of theories, operations on theories, and theory morphisms. The theoretical work performed under this contract has motivated the development and beginning implementation of the Specware system, which is our succesor to KIDS. Over the past few years we have focused on scheduling as an application domain.

### 1.1. Experimental Testbed – KIDS

KIDS [42, 33] is a program transformation system – one applies a sequence of consistency-preserving transformations to an initial specification and achieves a correct and efficient program. From the user's point of view the system allows the user to make high-level design decisions like, "design a divide-and-conquer algorithm for that specification" or "simplify that expression in context". We hope that decisions at this level will be both intuitive to the user and be high-level enough that useful programs can be derived within a reasonable number of steps.

The user typically goes through the following steps in using KIDS for program development.

1. *Develop a domain theory* – The user builds up a domain theory by defining appropriate types and functions. The user also provides laws that allow high-level reasoning about the defined functions. Our experience has been that distributive and monotonicity laws provide most of the laws that are needed to support design and optimization. Recently we have added a theory development component to KIDS that supports the automated derivation of distributive laws.

2. *Create a specification* – The user enters a specification stated in terms of the underlying domain theory.

3. *Apply a design tactic* – The user selects an algorithm design tactic from a menu and applies it to a specification. Currently KIDS has tactics for simple problem reduction (reducing a specification to a library routine) [30], divide-and-conquer [30], global search (binary search, backtrack, branch-and-bound) [32], and local search (hillclimbing) [20, 21].

4. *Apply optimizations* – The KIDS system allows the application of optimization techniques such as simplification, partial evaluation, finite differencing, and other transformations. The user selects an optimization method from a menu and applies it by pointing at a program expression. Each of the optimization methods are fully automatic and, with the exception of simplification (which is arbitrarily hard), take only a few seconds.

5. *Apply data type refinements* – The user can select implementations for the high-level data types in the program. Data type refinement rules carry out the details of constructing the implementation.

6. *Compile* – The resulting code is compiled to executable form. In a sense, KIDS can be regarded as a front-end to a conventonal compiler.

Actually, the user is free to apply any subset of the KIDS operations in any order – the above sequence is typical of our experiments in algorithm design.

## 1.2. Theory

Much of my work in the past few years has focused on automating the design of algorithms. I developed formal models, called *algorithm theories*, for various well-known classes of algorithms and developed formal/mechanizable design tactics for each class. Example classes were listed above.

More recent work has focused on theories and operations on theories as the formal underpinings of algorithm design as well as data structure design/refinement and general software development. Algorithm design is based on constructing a theory morphism[1] from an algorithm theory into a given application domain theory. Datatype design and and refinement are also based on constructing a theory morphism from one datatype theory into another. Generally, specifications are theories and the implementation of specifications is based on constructing a theory morphism into a (relatively) concrete, computationally-oriented theory. This formal view of software development has motivated research into the kinds of theories that are useful for specifying and reasoning about application domains and systems, as well as capturing knowledge about algorithms, data structures, and other kinds of programming knowledge. It has also led us to focus our attention on formal/automatable techniques for constructing theory morphisms.

### 1.3. Applications – Transportation Scheduling

We have used KIDS to derive extremely fast and accurate transportation schedulers from formal specifications. As test data we use strategic transportation plans which are generated by U.S. government planners. A typical problem, with 10,000 movement requirements, takes the derived scheduler 1 – 3 minutes to solve, compared with 2.5 *hours* for a deployed feasibility estimator (JFAST) and 36 *hours* for deployed schedulers (FLOGEN, ADANS). The computed schedules use relatively few resources and satisfy all specified constraints. The speed of this scheduler is due to the synthesis of strong constraint checking and constraint propagation code.

## 2. Significant Accomplishments

We briefly describe the main accomplishments of this project. Each topic is elaborated in subsequent sections.

1. *TPFDD Scheduling* – The U.S. Transportation Command and the component service commands use a relational database scheme called a TPFDD (Time-Phased Force and Deployment Data) for specifying the transportation requirements of an operation, such as Desert Storm or the Somalia relief effort. We developed a domain theory of TPFDD scheduling defining the concepts of this problem and developed laws for reasoning about them. KIDS (Kestrel Interactive Development System) was used to derive and optimize a variety of global search scheduling algorithms that perform constraint propagation [36]. The resulting code, generically called KTS (Kestrel Transportation Scheduler),

[1]A *theory morphism* from theory $A$ to theory $B$ is a translation of the language of $A$ to the language of $B$ such that theorems of $A$ translate to theorems of $B$.

has been run on a variety of TPFDDs generated by planners at USTRANSCOM and other sites.

We compared the performance of KTS with several other TPFDD scheduling systems, such as JFAST, FLOGEN, DITOPS, and PFE. We do not have access to JFAST and FLOGEN, but these are (or were) operational tools at AMC (Airlift Mobility Command, Scott AFB). According to [10] and David Brown (retired military planner consulting with the Planning Initiative), on a typical TPFDD of about 10,000 movement records, JFAST takes several hours and FLOGEN about 36 hours. KTS on a TPFDD of this size will produce a detailed schedule in *one to three minutes.* So KTS seems to be a factor of about 25 times faster than JFAST and over 250 times faster than FLOGEN. The currently operational ADANS system reportedly runs at about the same speed as FLOGEN. KTS is orders of magnitude faster than any other TPFDD scheduler known to us.

2. *Theater Scheduling*

   In 1994 we began to develop a scheduler to support PACAF (Pacific Air Force) at Hickham AFB, Honolulu which is tasked with in-theater scheduling of a fleet of 26 C-130 cargo aircraft in the Pacific region. We developed (and are continuing to evolve) a theory of theater transportation scheduling. Several variants of a theater scheduler (called ITAS for In-Theater Airlift Scheduler) have been developed to date, and more are planned. The interface to ITAS and integration with a commercial database package have been developed by BBN. ITAS runs on an Apple Powerbook laptop computer. The laptop platform makes it attractive both for field and command center operations. ITAS can currently produce ATOs (Air Tasking Orders) based on the schedules that it generates.

   The ITAS schedulers have emphasized flexibility and rich constraint modeling. Versions of ITAS were installed at PACAF in August 1994, September 1994, and February 1995. The most recent version simultaneously schedules the following classes of resources: (1) aircraft, (2) aircrews and their duty day cycles, (3) ground crews for unloading, and (4) ramp space at ports.

3. *Synthesis of constraint propagation code*

   A key technical achievement of this project was discovering and implementing technology for generating efficient constraint propagation code. The speed of the KTS schedulers derives from the extremely fast checking and propagation of constraint information at every node of the runtime search tree. Whereas some knowledge-based approaches to scheduling will search a tree at the rate of several nodes per second, our synthesized schedulers search *several hundred thousand* nodes per second.

   Briefly, the idea is to derive necessary conditions on feasibility of a candidate schedule. These conditions are called *cutting constraints.* The derived cutting constraints for a particular scheduling problem are analyzed to produce code that iteratively fixes violated constraints until the cutting constraints are satisfied. This iterative process subsumes the well-known processes of constraint propagation in the AI literature and the notion of cutting planes from the Operations Research literature [40, 46]. Constraint propagation is discussed in more detail in Section 4.2.3..

4. *Classification approach to design*

   We developed a new approach to the problem of how to construct refinements of specifications formally and incrementally. The idea is to use a taxonomy of abstract design concepts, each represented by a *design theory*. An abstract design concept is applied by constructing a specification morphism from its design theory to a requirement specification. Procedures for computing colimits and for constructing specification morphisms provide computational support for this approach. Although the classification approach applies to the incremental application of *any* kind of knowledge formally represented in a hierarchy of theories, our work mainly focused on a hierarchy of algorithm design theories and its applications to logistical applications [38, 37]. This technique enable us to integrate at a deep semantic level problem-solving methods from Computer Science (e.g. divide-and-conquer, global search), Artificial Intelligence (e.g. heuristic search, constraint propagation, neural nets), and Operations Research (e.g. Simplex, integer programming, network algorithms). Classification is discussed in more detail in Section 5..

5. *Derivation of Parallel Algorithms* – Although our algorithm design tactics have been used to produce algorithms with good parallel properties [31, 33], parallel algorithms has not been an explicit focus of our research. However for the purposes of the Workshop on Parallel Algorithm Derivation and Program Transformation held in New York City in 1991, I explored the application of the algorithm design tactics to some classic parallel sorting algorithms [39]. I choose Batcher's parallel mergesort algorithm which is quite difficult to explain and has some desirable features – it was the basis of one of the first sorting networks. The reason that the algorithm is so difficult to explain is that it is based on a shuffle constructor for sequences rather than the more familiar cons or concat: $\mathit{shuffle}([1,3,5],[2,4,6]) = [1,2,3,4,5,6]$, $\mathit{cons}(1,[2,3]) = [1,2,3]$, $\mathit{concat}([1,2,3],[4,5,6]) = [1,2,3,4,5,6]$. Since the constructors of a type have a heavy influence on the formulation of the operations and relations on a type, one would expect that a shuffle-based sequence theory would be quite different than cons-based theories. Indeed the hardest part of this derivation was to build up the shuffle-based theory of sequences – questions such as 'under what conditions is the shuffle of two sequences sorted?' lead to whole new concepts and the laws for reasoning about them. Once this theory is built up the derivation of a $O(\log n)$-time merge operation is straightforward and yields a $O(\log^2 n)$-time mergesort. Also derivable in this new shuffle-based theory is the well-known odd-even transposition sort and a variety of other shuffle-related algorithms.

6. *Problem Reduction Theories*

   Problem reduction is a pervasive technique for solving problems by reducing a problem instance to a structure of subproblem instances. Solutions to the subproblem instances are composed to form a solution to the initial instance. Divide-and-conquer is problem reduction used to find a single solution to a given problem. Problem reduction generators are used to enumerate all solutions. We produced an algorithm theory and design tactic for the class of problem reduction generators [34]. This algorithm theory, called *complete problem reduction theory*, provides the logical basis for dynamic programming, branch-and-bound, game-tree search, and other well-known algorithm paradigms. The design tactic is related to the divide-and-conquer tactic and results in the characteristic recurrence equation of problem reduction theory. If we decide to compute the characteristic recurrence equations from the bottom up, then we get the usual dynamic programming algorithms. Top-down control leads to branch-and-bound algorithms such as AO* and game-tree algorithms such as alpha-beta and SSS*.

This year we implemented in KIDS a new collection of tools based on theories, signature morphisms, theory morphisms, and theory translation. The tactics for divide-and-conquer and problem reduction generators were implemented using these tools and the result has been far more general and flexible tactics than previously. Partly this generality is due to the theories being indexed by a signature, as in algebraic theories. In previously published algorithm theories and design tactics we have had to fix the signature thereby limiting the applicability of the theory unnecessarily.

7. *Data Structure Design*

A subtype is often defined by a characteristic predicate over elements of a supertype. All operations on the subtype preserve this predicate as an invariant. Most interesting data structures are subtypes in this sense. For example, heaps are a subtype of binary trees that are characterized by the property that the children of a node are larger than the parent. From a design perspective, this suggests two key problems. First, how do the characteristic predicates arise during the development process. Must they simply be invented or are there formal techniques for deriving appropriate predicates? Second, given such a characteristic predicate, how can we formally develop a self-contained theory for the subtype – one without reference to operations of the parent type?

In this past year we have begun developing answers to these two questions. The first, how to derive the characteristic predicate of a subtype, has been explored via a detailed derivation of the classic heapsort algorithm. Heapsort is a specialized selection sort algorithm involving the use of the heap data structure and was the first known $O(n \log n)$ worst-case and average-case algorithm for sorting. Applying divide-and-conquer to the sorting problem (and choosing a simple compose) we derive a specification for the problem of selecting and extractingthe least element of a bag. This gives us an extended bag theory – the usual operations of equality, membership, construction, etc. augmented with min and extractmin. When we attempt to construct a theory morphism from this extended bag theory into binary tree theory, then unskolemization and various heuristically-guided choices result in the derivation of the heap property. The key choice is that the min operation (in bag theory) be translated as the root operation (in binary tree theory). This results in the heap property. There is a great deal more to explore in the theory of designing data structures, but it appears that the concepts and techniques that we have developed for designing algorithms provides many of the basic techniques.

The other question of how to develop a self-contained subtype theory has been adddressed in the following way. We developed a procedure that derives constructors for a subtype given constructors for the parent type and a characteristic predicate of the subtype. The procedure is sound in that if it terminates it produces a set of constructors that inductively generate the subset of elements of the parent type that satisfy the characteristic predicate. However the procedure need not terminate. In such cases the procedure seems to generate constructors for each value of the subtype – it does not find a recursive pattern that covers the type. For the other operators of the subtype theory we must design them so that they leave the subtype predicate invariant – often this is an algorithm design problem.

8. Software Evolution

   My colleague Y.V. Srinivas and I have begun to explore a formal approach to evolution of formal descriptions that is based on inference and dependency analysis. We view evolution as the transition from one consistent description to another. Each such transition can be decomposed into three phases: (1) start with a consistent description, (2) change some aspect of the description (possibly introducing inconsistency), (3) minimally change other parts of the description to re-establish consistency (change propagation). In general, change propagation is the maintenance of certain properties (such as consistency, well-formedness, etc.) while changing others.

   To make the problem of change propagation tractable, we restrict our attention to changes which are monotonic, i.e., generalizations or specializations (other changes can be represented as combinations of these). Using an explicit representation of a consistency property as a formula, we use a special form of inference, *directed inference*, to determine which parts of the description to change in order to re-establish the desired consistency property. The inference makes use of dependency information which indicates the direction and amount of change, *variance*, of each entity in the domain with respect to changes in other entities.

9. *KIDS and Challenge Problems* – In 1991 we applied KIDS to derive a somewhat larger problem than before [35]. We obtained a structured English specification of the Track Assignment portion of an Air Traffic Control system that was developed by Hughes Aircraft. This was about 6 pages of text. A domain theory was developed in KIDS (about 24 pages of formal text) in which most of the laws and rules were automatically derived using the theory development mode in KIDS. Then various design, optimization, and refinement operations were applied to generate code from a specification of a track assignment module. This code was similar in structure to hand generated ADA produced by Hughes personnel. The main difference was that our code was expressed in a functional subset of the Regroup language (our local extension of the Refine language).

## 3. Specifying a Scheduler

### 3.1. What is Scheduling?

The essential notion of scheduling is that certain activities are assigned to resources over certain time intervals. Various constraints on the assignments must be satisfied and certain measures of the cost or "goodness" of the assignment are to be optimized.

A domain theory for scheduling defines the basic concepts of scheduling and the laws for reasoning about the concepts. After a review of the relevant literature (e.g. [11]) we have identified the following general components of a scheduling domain theory.

1. *Activities* – A model of the activities can include their internal structure and characteristics, hierarchies of activity abstractions, and various operations on activities.
2. *Resources* – A model of the resources can include their internal structure and characteristics, hierarchies of resource abstractions, and various operations on resources.
3. *Time* – A time model can include a calculus of time-points or time-intervals [1, 19].
4. *Constraints* – A constraint model includes the language for stating constraints and a calculus for reasoning about them. Several classes of constraints commonly arise in practice. The most common are *precedence constraints* (which state that one activity must precede another) and *capacity constraints* (which state bounds on the capacities of resources). A constraint calculus is used to analyze constraints and to propagate the effects of new constraints through a given constraint set. Fox et al. also identify physical constraints, organizational constraints, preferences, enablement constraints, and availability constraints.
5. *Objectives* – Typically we seek to minimize the cost of a schedule. Cost can be measured in terms of time to completion, work-in-progress, total cost of consumed resources, and so on.
6. *Scheduling problem* – Using the above concepts we can formulate a variety of scheduling problems. A reservation is a triple consisting of an activity, a resource, and a time interval. Generally, a schedule is a set of *reservations* that satisfy a collection of constraints and optimize (or produce a reasonably good value of) the objective.

$$\{ \langle \mathit{activity},\ \mathit{resource},\ \mathit{time\ interval} \rangle \mid \mathit{constraints} \}.$$

Many scheduling problem intancess are overconstrained – there are too few resources to schedule the activities and satisfy all constraints. Usually overconstrained problems are dealt with by relaxing the constraints and trying to satisfy as many of the constraints as possible. The usual method is to move constraints into the objective function. This entails reformulating the constraint so that it yields a quantitative measure of how well it has been satisfied. See further discussion in Section 4.2.4..

## 3.2. Strategic Transportation Scheduling

Transportation scheduling specializes the above general notion of scheduling: activities correspond to *movement requirements* and resources correspond to transportation assets such as planes, ships, and trucks.

A typical movement requirement has the following information

| | | |
|---|---|---|
| *move–type* : *movement–type* | $\mapsto$ | *BULK–MOVEMENT* |
| *quantity* : *integer* | $\mapsto$ | 2 (STONS - Short TONS) |
| *release–date* : *time* | $\mapsto$ | 0 (seconds from C-date) |
| *due–date* : *time* | $\mapsto$ | 86400 (seconds from C-date) |
| *poe* : *port* | $\mapsto$ | *UHHZ* |
| *pod* : *port* | $\mapsto$ | *VRJT* |
| *distance* : *integer* | $\mapsto$ | 5340 (nautical miles) |
| *mode* : *symbol* | $\mapsto$ | *AIR* |

Here quantity for AIR movements is in short tons (STONs); the release and due dates are in seconds starting from C-DATE; poe (port of embarkation) and pod (port of debarkation) are given by code names; distance is in nautical miles, and the transportation mode is either AIR or SEA. A collection of movement requirements is called a TPFDD (Time-Phased Force Deployment Data).

Resources are characterized by their capacities (both passenger (PAX) and cargo capacities), and travel rate in knots.

As an example, we used a small dataset extracted from a TUNISIA TPFDD created at AFSC. This problem instance involves 480 movement requirements from 20 airports and 3 seaports to 8 airports and 2 seaports. Available air resources include KC10s, C-141s, C-5s and sea resources include tankers (small, medium, and large), RO-ROs, LASHs, sea barges, containerships, and breakbulks.

Eleven constraints characterize a feasible schedule for a simple TPFDD problem:

1. *Consistent POE and POD* – The POE and POD of each movement requirement on a given trip of a resource must be the same.

2. *Consistent Resource Class* – Each resource can handle only some movement types. For example, a C-141 can handle bulk and oversize movements, but not outsize movements.

3. *Consistent PAX and Cargo Capacity* – The capacity of each resource cannot be exceeded.

4. *Consistent Release Time* – The start time of a movement (its Available to Load Date (ALD)) must not precede its release time.

5. *Consistent Arrival time* – The finish time of a trip must not precede the Earliest Arrival Date (EAD) of any of the transported movement requirements.

6. *Consistent Due time* – The finish time of a movement (its Latest Arrival Date (LAD)) must not be later than its due time.

7. *Consistent Trip Separation* – Movements scheduled on the same resource must start either simultaneously or with enough separation to allow for return trips. The inherently disjunctive and relative nature of this constraint makes it more difficult to satisfy than the others.

8. *Consistent Resource Use* – Only the given resources are used.

9. *Completeness* – All movement requirements must be scheduled.

In the next section we discuss the formalization of the above concepts. This problem does not consider certain aspects of transportation scheduling, such as aircrew scheduling, ground crew scheduling, maintenance, resource utilization rates, load/unload rates, port characteristics, etc. Each of these problem features have been handled in various more elaborate specifications.

## 3.3. (Re-)Formulating Domain Theories for Transportation Scheduling

In the most general view, scheduling is the construction of a set of reservations that satisfy given feasibility constraints and achieve "good" values of an objective function. Formally, the schedule is a relation, or even a simple relational database. A formal domain theory based on this view is given in Appendix A in [41]. The theory provides precise definitions for the concepts, constraints, objectives, and laws used to model this application domain.

This relational view however is not always the most efficient for particular problems. We may be able to reformulate the problem, incorporating constraints and objectives, yielding a problem statement that is more amenable to efficient problem-solving. In the following we present a series of transformations that reformulate the domain theory.

In most transportation problems, each movement requirement corresponds to a unique reservation – it is scheduled exactly once with a unique resource and start time. We can make this functional dependence explicit by treating a schedule as a *map* from movement requirements to resource/time tuples. In Figure 1 we show the effect of this reformulation on the schedule datatype.

Next the trip separation constraint suggests that this map is many-to-one, since several movements can take place simultaneously on the same resource. Inverting the map will induxe a partition on movement requirements. In terms of the transportation domain, inverting the map will make simultaneous movements explicit and thereby introducing the concept of a *trip* and the *manifest* of a trip.

Next we notice that the domain of a schedule map is a product of two types and these types have quite different properties (algebras): resources are a discrete set and time is (effectively) continuous and linear. The linear nature of time can be exploited by currying (to separate the two domain datatypes) and transforming the submap (from time to manifest) to a

$$\begin{tikzcd}
\{ \langle \textit{movement-record},\ \textit{resource},\ \textit{start-time} \rangle \} \arrow[d, "\textbf{reify the functional dependence}"] & \textbf{a schedule represented as a set of reservations} \\
\{ \textit{movement-record} \longmapsto \langle \textit{resource},\ \textit{start-time} \rangle \} \arrow[d, "\textbf{invert the map}"] & \\
\{ \langle \textit{resource},\ \textit{start-time} \rangle \longmapsto \{ \textit{movement-record} \} \} \arrow[d, "\textbf{Curry}"] & \textbf{the notion of "trip" and "manifest" introduced} \\
\{ \textit{resource} \longmapsto \{ \textit{start-time} \longmapsto \{ \textit{movement-record} \} \} \} \arrow[d, "\textbf{exploit linear order of time}"] & \\
\{ \textit{resource} \longmapsto [ \langle \textit{start-time},\ \{ \textit{movement-record} \} \rangle ] \} &
\end{tikzcd}$$

Figure 1: Reformulating a Scheduling Specification

sequence, thereby making the linear structure of time explicit and introducing the concept of an *itinerary.*

This series of reformulations has dramatic effect on the trip separation constraint. In the initial formulation (in terms of reservations) this constraint involves $O(n^2)$ binary constraints between the $n$ movements scheduled on a given resource. In the final formulation (in terms of a linearized inverse map) this constraint is reduced to $O(n)$ binary constraints between the start times of consecutive trips.

For example, on a transportation problem involving over 15,000 movement requirements obtained from the U.S. Transportation Command, the scheduler produces a complete feasible schedule in about five minutes. A straightforward constraint network formulation based on this problem data would have over 31,000 variables and 120,125,000 constraints. Incorporating some of the structure of the problem, such as the linearity of time, allows reformulating this to a system of about 108,700 constraints. However, this is still a such large formulation that it seems an implicit representation is necessary to find feasible schedules efficiently.

The final reformulation is given in Appendix B in [41] and is the theory actually used to derive a scheduler.

## 3.4. Formal Specification of a Scheduler

The informal specification above can be expressed as follows:

```
function TS
    (mvrs : seq(movement-record),
     assets : seq(resource-name))
returns (sched : map(resource-name, seq(trip)) |
            Consistent-POE(sched)
            ∧ Consistent-POD(sched)
            ∧ Consistent-Release-Times(sched)
            ∧ Consistent-Arrival-Times(sched)
            ∧ Consistent-Due-Times(sched)
            ∧ Consistent-Trip-Separation(sched)
            ∧ Consistent-Pax-Resource-Capacity(sched)
            ∧ Consistent-Cargo-Resource-Capacity(sched)
            ∧ Consistent-Movement-Type-and-Resource(sched)
            ∧ Available-Resources-Used(assets, sched)
            ∧ Scheduled-mvrs(sched) = seq-to-set(mvrs))
```

This specifies a function called *TS* that takes two inputs, a sequence of movement records called *mvrs* and a sequence of resources called *assets*. The function returns a schedule, which has type *map(resource-name, seq(trip))* and must satisfy the 11 conjoined constraints. Each constraint is defined in the domain theory; for example:

```
function $CONSISTENT\text{-}DUE\text{-}TIMES$
    $(sched : schedule) : boolean$
    $= \forall(rsrc : resource\text{-}name,\ trp : integer,\ mvr : movement\text{-}record)$
            $(rsrc \in domain(sched)$
             $\wedge\ trp \in [1..size(sched(rsrc))]$
             $\wedge\ mvr \in sched(rsrc)(trp).manifest$
             $\Longrightarrow$
            $sched(rsrc)(trp).start\text{-}time$
            $\leq (mvr.due\text{-}date - sched(rsrc)(trp).trip\text{-}duration)$
```

This predicate expresses the constraint that every scheduled movement-record arrives before its due date.

# 4. Synthesizing a Scheduler

## 4.1. Approach

### 4.1.1. Problem Theories

We briefly review some basic concepts from algebra and logic. A *theory* is a structure $\langle S, \Sigma, A\rangle$ consisting of a set of sort symbols $S$, operations over those sorts $\Sigma$, and axioms $A$ to constrain the meaning of the operations. A *theory morphism* (*theory interpretation*) maps from the sorts and operations of one theory to the sorts and expressions over the operations of another theory such that the image of each source theory axiom is valid in the target theory. A *parameterized theory* has formal parameters that are themselves theories [14]. The binding of actual values to formal parameters is accomplished by a theory morphism. Theory $\mathcal{T}_2 = \langle S_2, \Sigma_2, A_2\rangle$ *extends* (or is an *extension* of) theory $\mathcal{T}_1 = \langle S_1, \Sigma_1, A_1\rangle$ if $S_1 \subseteq S_2$, $\Sigma_1 \subseteq \Sigma_2$, and $A_1 \subseteq A_2$.

Problem theories define a problem by specifying a domain of problem instances or inputs and the notion of what constitutes a solution to a given problem instance. Formally, a *problem theory* $\mathcal{B}$ has the following structure.

| | |
|---|---|
| **Sorts** | $D, R$ |
| **Operations** | $I : D \rightarrow Boolean$ |
| | $O : D \times R \rightarrow Boolean$ |

The *input condition* $I(x)$ constrains the input domain $D$. The *output condition* $O(x, z)$ describes the conditions under which output domain value $z \in R$ is a *feasible solution* with respect to input $x \in D$. Theories of booleans and sets are implicitly imported. Problems of finding optimal feasible solutions can be treated as extensions of problem theory by adding a cost domain, cost function, and ordering on the cost domain.

For example, the problem of finding feasible schedules can be presented as a problem theory via a theory interpretation into the domain theory of transportation scheduling:[2]

$$
\begin{array}{lll}
D & \mapsto & seq(movement\text{–}record) \times seq(resource) \\
I & \mapsto & \lambda(Mvrs, resources)\ true \\
R & \mapsto & map(resource, seq(trip)) \\
O & \mapsto & \lambda(Mvrs, resources, sched) \\
 & & \quad Consistent\text{–}POE(sched) \\
 & & \quad \wedge\ Consistent\text{–}POD(sched) \\
 & & \quad \wedge\ Consistent\text{–}Release\text{–}Times(sched) \\
 & & \quad \wedge\ Consistent\text{–}Arrival\text{–}Times(sched) \\
 & & \quad \wedge\ Consistent\text{–}Due\text{–}Times(sched) \\
 & & \quad \wedge\ Consistent\text{–}Trip\text{–}Separation(sched) \\
 & & \quad \wedge\ Consistent\text{–}Pax\text{–}Resource\text{–}Capacity(sched) \\
 & & \quad \wedge\ Consistent\text{–}Cargo\text{–}Resource\text{–}Capacity(sched) \\
 & & \quad \wedge\ Consistent\text{–}Movement\text{–}Type\text{–}and\text{–}Resource(sched) \\
 & & \quad \wedge\ Available\text{–}Resources\text{–}Used(resources, sched) \\
 & & \quad \wedge\ Scheduled\text{–}mvrs(sched) = seq\text{–}to\text{–}set(mvrs)
\end{array}
$$

### 4.1.2. Algorithm Theories

An *algorithm theory* represents the essential structure of a certain class of algorithms $\mathcal{A}$ [43]. Algorithm theory $\mathcal{A}$ extends problem theory $\mathcal{B}$ with any additional sorts, operators, and axioms needed to support the correct construction of an $\mathcal{A}$ algorithm for $\mathcal{B}$. A theory morphism from the algorithm theory into some problem domain theory provides the problem-specific concepts needed to construct an instance of an $\mathcal{A}$ algorithm.

For example, global search theory (presented below in Section 6.2.1) extends problem theory with the basic concepts of backtracking: subspace descriptors, initial space, the splitting and extraction operations, filters, and so on. A divide-and-conquer theory would extend problem theory with concepts such as decomposition operators and composition operators [30, 34].

## 4.2. Synthesizing a Scheduler

There are two basic approaches to computing a schedule: local and global. Local methods focus on individual schedules and similarity relationships between them. Once an initial schedule is obtained, it is iteratively improved by moving to neighboring structurally similar

[2]The domain theory includes definitions for the types of movement-record, resource, trip (a record comprised of start-time and manifest), and schedule (a map from resource to sequence of trip).

schedules. Repair strategies [52, 22, 3, 28], and fixpoint iteration [7], and linear programming algorithms are examples of local methods.

Global methods focus on sets of schedules. A feasible or optimal schedule is found by repeatedly splitting an initial set of schedules into subsets until a feasible or optimal schedule can be easily extracted. Backtrack, heuristic search, and branch-and-bound methods are all examples of global methods. We explore the application of global methods. In the following subsections we formalize the notion of global search method and show how it can be applied to synthesize a scheduler. Other projects taking a global approach include ISIS [12], OPIS/DITOPS [47], and MicroBoss [25] (all at CMU).

### 4.2.1. Global Search Theory

The basic idea of global search is to represent and manipulate sets of candidate solutions. The principal operations are to *extract* candidate solutions from a set and to *split* a set into subsets. Derived operations include various *filters* which are used to eliminate sets containing no feasible or optimal solutions. Global search algorithms work as follows: starting from an initial set that contains all solutions to the given problem instance, the algorithm repeatedly extracts solutions, splits sets, and eliminates sets via filters until no sets remain to be split. The process is often described as a tree (or DAG) search in which a node represents a set of candidates and an arc represents the split relationship between set and subset. The filters serve to prune off branches of the tree that cannot lead to solutions.

The sets of candidate solutions are often infinite and even when finite they are rarely represented extensionally. Thus global search algorithms are based on an abstract data type of intensional representations called *space descriptors* (denoted by hatted symbols). In addition to the extraction and splitting operations mentioned above, the type also includes a predicate *satisfies* that determines when a candidate solution is in the set denoted by a descriptor. Further, there is a refinement relation on spaces that corresponds to the subset relation on the sets denoted by a pair of descriptors.

The various operations in the abstract data type of space descriptors together with problem specification can be packaged together as a theory. Formally, abstract *global search theory* (or simply *gs–theory*) $\mathcal{G}$ is presented in Figure 2, where $D$ is the input domain, $R$ is the output domain, $I$ is the input condition, $O$ is the output condition, $\hat{R}$ is the type of space descriptors, $\hat{I}$ defines legal space descriptors, $\hat{r}$ and $\hat{s}$ vary over descriptors, $top(x)$ is the descriptor of the initial set of candidate solutions, $Satisfies(z, \hat{r})$ means that $z$ is in the set denoted by descriptor $\hat{r}$ or that $z$ satisfies the constraints that $\hat{r}$ represents, and $Extract(z, \hat{r})$ means that $z$ is directly extractable from $\hat{r}$.

The relations *Split–Arg* and *Split–Constraint* are used to determine and perform splitting. In particular, if $Split\text{–}Arg(x, \hat{r}, c)$ then $c$ is information that characterizes (or informs) one branch of the split. $Split\text{–}Constraint(x, \hat{r}, c, \hat{s})$ means that $\hat{s}$ results from incorporating information $c$ into the descriptor $\hat{r}$ (with respect to input $x$). *Split–Arg* is used to control the generation of children of a node in the search tree and *Split–Constraint* is used to

**Spec** *Global-Search*

| | | |
|---|---|---|
| **Sorts** | $D$ | *input domain* |
| | $R$ | *output domain* |
| | $\hat{R}$ | *subspace descriptors* |
| | $\hat{C}$ | *splitting information* |

**Operations**

| | |
|---|---|
| $I : D \rightarrow boolean$ | *input condition* |
| $O : D \times R \rightarrow boolean$ | *input/output condition* |
| $\hat{I} : D \times \hat{R} \rightarrow boolean$ | *subspace descriptors condition* |
| $Satisfies : R \times \hat{R} \rightarrow boolean$ | *denotation of descriptors* |
| $Split\text{--}Arg : D \times \hat{C} \times \hat{R} \rightarrow boolean$ | *specifies arguments to split constraint* |
| $Split\text{--}Constraint : D \times \hat{R} \times \hat{C} \times \hat{R} \rightarrow boolean$ | *parameterized splitting constraint* |
| $Extract : R \times \hat{R} \rightarrow boolean$ | *extractor of solutions from spaces* |
| $\Psi : D \times R \times \hat{R} \rightarrow boolean$ | *cutting constraint* |
| $\xi : D \times \hat{R} \rightarrow boolean$ | *cutting constraint* |
| $\sqsupseteq : D \times \hat{R} \times \hat{R} \rightarrow boolean$ | *refinement relation* |
| $top : D \rightarrow \hat{R}$ | *initial space* |
| $bot : \hat{R}$ | *inconsistent space* |

**Axioms**

GS0. All feasible solutions are in the *top* space

$$I(x) \wedge O(x,z) \implies Satisfies(z, top(x))$$

GS1. All solutions in a space are finitely extractable

$$I(x) \wedge \hat{I}(x,\hat{r})$$
$$\implies (Satisfies(z,\hat{r}) \iff \exists(\hat{s})\ (\ Split^{*}(x,\hat{r},\hat{s}) \wedge Extract(z,\hat{s})))$$

GS2. Specification of Cutting Constraint

$$Satisfies(z,\hat{r}) \wedge O(x,z) \implies \Psi(x,z,\hat{r})$$

GS3. Definition of Cutting Constraint on Spaces

$$\xi(x,\hat{r}) \iff \forall(z:R)(\ Sat(z,\hat{r}) \implies \Psi(x,z,\hat{r}))$$

GS4. Definition of Refinement

$$\hat{r} \sqsupseteq \hat{s} \iff \forall(z:R)(Satisfies(z,\hat{s}) \implies Satisfies(z,\hat{r}))$$

GS5. $\langle \hat{R}, \sqsupseteq, \sqcap, top, bot \rangle$ is a bounded meet-semilattice with *bot* as universal lower bound.

**end spec**

Figure 2: Global Search Theory

specify one child. *Split–Constraint* can be thought of as a parameterized constraint whose alternative arguments are supplied by *Split–Arg*.

The refinement relation $\hat{r} \sqsupseteq \hat{s}$ holds when $\hat{s}$ denotes a subset of the set denoted by $\hat{r}$. Further, $\hat{R}$ together with $\sqsupseteq$ forms a bounded semilattice. This structure will play a crucial role in constraint propagation algorithms.

Note that all variables in the axioms are assumed to be universally quantified unless explicitly specified otherwise. Axiom GS0 asserts that the initial descriptor $\hat{r}_0(x)$ is a legal descriptor. Axiom GS1 asserts that legal descriptors split into legal descriptors and that *Split* induces a well-founded ordering on spaces. Axiom GS2 constrains the denotation of the initial descriptor — all feasible solutions are contained in the initial space. Axiom GS3 gives the denotation of an arbitrary descriptor $\hat{r}$ — an output object $z$ is in the set denoted by $\hat{r}$ if and only if $z$ can be extracted after finitely many applications of *Split* to $\hat{r}$ where

$$Split^*(x, \hat{r}, \hat{s}) \iff \exists(k : Nat)\ \ Split^k(x, \hat{r}, \hat{s})$$

and

$$Split^0(x, \hat{r}, \hat{t}) \iff \hat{r} = \hat{t}$$

and for all natural numbers $k$

$$Split^{k+1}(x, \hat{r}, \hat{t})$$

$$\iff \exists(\hat{s} : \hat{R},\ i : \hat{C})\ (\ Split\text{–}Arg(x, \hat{r}, i) \wedge Split\text{–}Constraint(x, \hat{r}, i, \hat{s}) \wedge Split^k(x, \hat{s}, \hat{t})).$$

Axiom GS4 asserts that if $\hat{r}$ splits to $\hat{s}$ then $\hat{r}$ also refines to $\hat{s}$; thus the refinement relation on $\hat{R}$ is weaker than the split relation. We also need the axioms that $\langle \hat{R}, \sqsupseteq, \sqcap \rangle$ is a semilattice. For simplicity, we write $\hat{r} \sqsupseteq \hat{s}$ rather than the correct $\sqsupseteq (x, \hat{r}, \hat{s})$; and similarly $\hat{r} \sqcap \hat{s}$.

For example, a simple global search theory of scheduling has the following form. Schedules are represented as maps from resources to sequences of trips, where each trip includes earliest-start-time, latest-start-time, port of embarkation, port of debarkation, and manifest (set of movement records or ULNs + CINs + PINs from the TPFDD). The type of schedules has the invariant (or subtype characteristic) that for each trip, the earliest-start-time is no later than the latest-start-time. A partial schedule is a schedule over a subset of the given movement records.

The initial (partial) schedule is just the empty schedule – a map from the available resources to the empty sequence of trips. A partial schedule is extended by first selecting a movement record *mvr* to schedule, then selecting a resource $r$, and then a trip $t$ on $r$ (either an existing trip or a newly created one) – the triple $\langle mvr, r, t\rangle$ constitutes the information $c$ of *Split–Arg*. *Split–Constraint* given $\langle mvr, r, t\rangle$ creates an extended schedule that has *mvr* added to the manifest of trip $t$ on resource $r$. The alternative ways that a partial schedule can be extended naturally gives rise to the branching structure underlying global search algorithms.

The formal version of this global search theory of scheduling can be inspected in the domain theory in Appendix C in [41].

### 4.2.2. Pruning Mechanisms

When a partial schedule is extended it is possible that some problem constraints are violated in such a way that further extension to a complete feasible schedule is impossible. In tree search algorithms it is crucial to detect such violations as early as possible.

*Pruning* tests are derived in the following way. The test

$$\exists(z)\ (Satisfies(z,\hat{r})\ \land\ O(x,z)) \tag{1}$$

decides whether there exist any feasible solutions that are in the space denoted by $\hat{r}$. If we could decide this at each node of our branching structure then we would have perfect search – no deadend branches would ever be explored. In practice it would be impossible or horribly complex to compute (1), so we rely instead on an inexpensive approximation to it. In fact, if we approximate (1) by weakening it (deriving a necessary condition of it) we obtain a sound pruning test. That is, suppose we can derive a test $\Phi(x,\hat{r})$ such that

$$\exists(sched)\ (Satisfies(z,\hat{r})\ \land\ O(x,z)) \implies \Phi(x,\hat{r}). \tag{2}$$

By the contrapositive of (2), if $\neg\Phi(x,\hat{r})$ then there are no feasible solutions in $\hat{r}$, so we can eliminate it from further processing. A global search algorithm will test $\Phi$ at each node it explores, pruning those nodes where the test fails.

More generally, necessary conditions on the existence of feasible (or optimal) solutions below a node in a branching structure underlie pruning in backtracking and the bounding and dominance tests of branch-and-bound algorithms [32].

It appears that the bottleneck analysis advocated in the constraint-directed search projects at CMU [11, 25] leads to a semantic approximation to (1), but neither a necessary nor sufficient condition. Such a *heuristic* evaluation of a node is inherently fallible, but if the approximation is close enough it can provide good search control with relatively little backtracking.

To derive pruning tests for the strategic transportation scheduling problem, we instantiate (1) with our definition of *Satisfies* and *O* and use an inference system to derive necessary conditions. The resulting tests are fairly straightforward; of the 11 original feasibility constraints, 7 yield pruning tests on partial schedules. For example, the partial schedule must satisfy *Consistent-POE*, *Consistent-POD*, *Consistent-Pax-Resource-Capacity*, *Consistent-Cargo-Resource-Capacity*, *Consistent-Movement-Type-and-Resource*, and *Available-Resources-Used*. The reader may note that computing these tests on partial schedules is rather expensive and mostly unnecessary – later program optimization steps will however reduce these tests to fast and irredundant form. For example, the first test will reduce to checking that when we place a movement record *mvr* on trip *t*, we check that the POE of *mvr* and *t* are consistent.

For details of deriving pruning mechanisms for other problems see [32, 42, 43, 33].

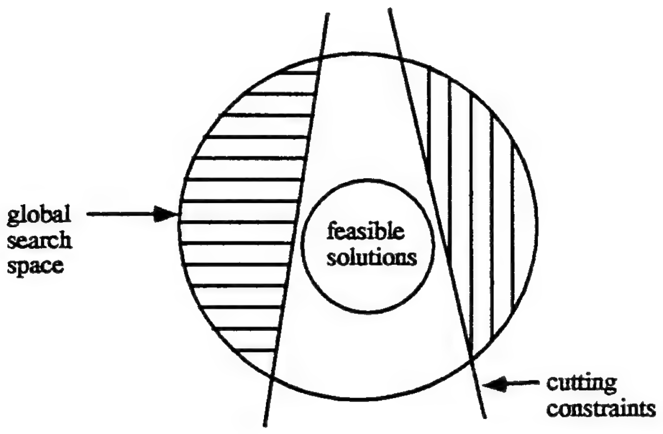

Figure 3: Global Search Subspace and Cutting Constraints

### 4.2.3. Cutting Constraints and Constraint Propagation

Constraint propagation is a more general technique that is crucial for early detection of infeasibility. We developed a general mechanism for deriving constraint propagation code and applied it to scheduling.

Each node in a backtrack tree can be viewed as a data structure that denotes a set of candidate solutions – in particular the solutions that occur in the subtree rooted at the node (see Figure 3). Thus the root denotes the set of all candidate solutions found in the tree.

Pruning has the effect of removing a node (set of solutions) from further consideration. In contrast, constraint propagation has the effect of changing the space descriptor so that it denotes a smaller set of candidate solutions. The effect of constraint propagation is to spread information through the subspace descriptor resulting in a tighter descriptor and possibly exposing infeasibility. Pruning can be treated as a special case of propagation in which a space is refined to descriptor that denotes the empty set of solutions.

Constraint propagation is based on the notion of *cutting constraints* which are necessary conditions $\Psi(x, z, \hat{r})$ that a candidate solution $z$ satisfying $\hat{r}$ is feasible:

$$\forall(x : D, \hat{r} : \hat{R},\ z : R)(\mathit{Satisfies}(z, \hat{r}) \ \wedge\ O(x, z) \implies \Psi(x, z, \hat{r})) \tag{3}$$

See Figures 3 and 4. In order to get a test on spaces that decides whether $\Psi$ has been incorporated, we make one further definition:

$$\xi(x, \hat{r}) \iff \forall(z : R)(\mathit{Satisfies}(z, \hat{r}) \implies \Psi(x, z, \hat{r})) \tag{4}$$

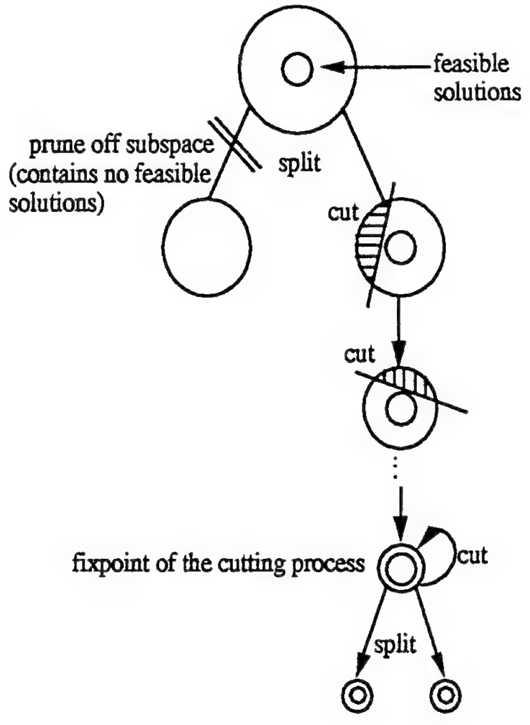
feasible solutions
prune off subspace (contains no feasible solutions)
split
cut
cut
fixpoint of the cutting process
cut
split

Figure 4: Pruning and Constraint Propagation

The test $\xi(x,\hat{r})$ holds exactly when all candidate solutions in $\hat{r}$ satisfy $\Psi$, and we say that $\hat{r}$ *satisfies* $\xi$.

The key question at this point is: Given a descriptor $\hat{r}$ that doesn't satisfy $\xi$, how can we incorporate $\xi$ into $\hat{r}$? The answer is to find the greatest refinement of $\hat{r}$ that satisfies $\xi$; we say $\hat{t}$ *incorporates* $\xi$ into $\hat{r}$ if

$$\hat{t} = max_{\sqsupseteq}\{\hat{s} \mid \hat{r} \sqsupseteq \hat{s} \ \wedge \ \xi(x,\hat{s})\}. \tag{5}$$

which asserts that $\hat{t}$ is maximal over the set of descriptors that refine $\hat{s}$ and satisfy $\xi$, with respect to ordering $\sqsupseteq$. We want $\hat{t}$ to be a refinement of $\hat{r}$ so that all of the information in $\hat{r}$ is preserved and we want $\hat{t}$ to be maximal so that no other information than $\hat{r}$ and $\xi$ is incorporated into $\hat{t}$.

The next question concerns the conditions under which Formula (5) is satisfiable. Assuming that $\hat{R}$ is a semilattice, we can use variants of Tarski's fixpoint theorem (c.f. [7]):

**Theorem** If there is a function $f$ such that

1. $f$ is monotonic on $\hat{R}$ (i.e. $\hat{s} \sqsupseteq \hat{t} \implies f(x,\hat{s}) \sqsupseteq f(x,\hat{t})$)
2. $f$ is deflationary (i.e. $\hat{r} \sqsupseteq f(x,\hat{r})$)

3. $f$ has fixed-points satisfying $\xi$ (i.e. $f(x,\hat{r}) = \hat{r} \iff \xi(x,\hat{r})$)

then (1) $\hat{t} = max_{\sqsupseteq}\{\hat{s} \mid \hat{r} \sqsupseteq \hat{s} \ \wedge \ \xi(x,\hat{s})\}$ exists
and (2) $\hat{t}$ is the greatest fixpoint of $f$; i.e. $\hat{t}$ can be computed by iteratively applying $f$ to $\hat{r}$ until a fixpoint is reached.

The challenge is to construct a monotonic, deflationary function whose fixed-points satisfy $\xi$. A general construction in terms of global search theory can be sketched as follows. Let

$$f(x,\hat{r}) = \begin{cases} \hat{r} & \textit{if } \xi(x,\hat{r}) \\ \ldots & \textit{if } \neg\xi(x,\hat{r}) \end{cases}$$

The intent is to define $f$ so that it has fixpoints exactly when $\xi(x,\hat{r})$ holds. When $\xi(x,\hat{r})$ doesn't hold, then we know (by the definition of $\xi$ and the contrapositive of formula (3)) that

$$\exists(z : R)(\mathit{Satisfies}(z,\hat{r}) \ \wedge \ \neg O(x,z))$$

i.e. there are some infeasible solutions in the space described by $\hat{r}$. Ideally $\neg\xi(x,\hat{r})$ is a *constructive* assertion, so it provides information on *which* solutions are infeasible and how to eliminate them. In place of the ellipsis above we require a new descriptor that refines $\hat{r}$ (so $f$ is decreasing on all inputs), allows $f$ to be monotone, and eliminates some of the infeasible solutions indicated by $\neg\xi(x,\hat{r})$. In general it is difficult to see how to achieve this end without assuming special structure to $\hat{R}$ and $\xi$.

We have identified some special cases for which an analytic procedure can produce the necessary iteration function $f$ from $\xi$. These special cases subsume our scheduling applications and many related Constraint Satisfaction Problems (CSP) problems. Suppose that the constraint $\xi$ has the form

$$B(x,\hat{r}) \sqsupseteq \hat{r} \tag{6}$$

where $B(x,\hat{r})$ is monotonic in $\hat{r}$. We say that $\xi$ is a *Horn-like constraint* by generalization of Horn clauses in logic. Notice that the occurrence of $\hat{r}$ on the right-hand side of the inequality has positive polarity (i.e. it is monotonic in $\hat{r}$), whereas the occurrence(s) of $\hat{r}$ on the left-hand side have negative polarity (i.e. are antimonotonic). If the constraint were boolean (with $B$ and $\hat{r}$ being boolean values and $\sqsupseteq$ being implication), then this would be called a definite Horn clause. When our constraints are Horn-like, then there is a simple definition for the desired function $f$:

$$f(x,\hat{r}) = \begin{cases} \hat{r} & \textit{if } B(x,\hat{r}) \sqsupseteq \hat{r} \\ B(x,\hat{r}) \sqcap \hat{r} & \textit{if } \neg B(x,\hat{r}) \sqsupseteq \hat{r} \end{cases}$$

or equivalently

$$f(x,\hat{r}) = B(x,\hat{r}) \sqcap \hat{r}.$$

It is easy to check that $f$ is monotone in $\hat{r}$, deflationary, and has fixed-points exactly when $\xi$ holds. Therefore, simple iteration of $f$ will converge to the descriptor that incorporates

$\xi$ into $\hat{r}$. However, if $\hat{r}$ is an aggregate structure such as a tuple or map, then the changes made at each iteration may be relatively sparse, so the simple iteration approach may be grossly inefficient. We found this feature to be characteristic of scheduling and other CSPs. Our approach to solving this problem is to focus on single point changes and to exploit dependence analysis. For each component of $\hat{r}$ we define a separate change propagation procedure. The arguments to a propagation procedure specify a change to the component. This change is performed and then the change procedures for all other components that could be affected by the change are invoked. Static dependence analysis at design-time is used to determine which constraints could be affected by a change to a given component.

A program scheme for global search with constraint propagation is presented in Figure 5. The global search design tactic in KIDS first instantiates this scheme, then invokes a tactic for synthesizing propagation code to satisfy the specification *F–split–and–propagate*.

*CSPs with Horn-like constraints*

We now elaborate the previous exposition of propagation of Horn-like constraints arising in CSPs. To keep matters simple, yet general, suppose that the output datatype $R$ is *map(VAR, VALSET)*, where $VAR$ is a type of variables, and $VALSET$ is a type that denotes a set of values (this implies that all the variables have the same type and refinement ordering), and the $\sqsupseteq$ relation has the form:

$$\hat{r} \sqsupseteq \hat{s} \;\; \textit{iff} \;\; \bigwedge_{v} \hat{r}(v) \sqsupseteq \hat{s}(v).$$

Suppose further that $\xi$ is a conjunction of constraints giving bounds on the variables:

$$\xi(x, \hat{r}) \iff \bigwedge_{v} B_v(x, \hat{r}) \sqsupseteq \hat{r}(v)$$

where $B_v(x, \hat{r})$ is monotonic in $\hat{r}$. Under these assumptions, $\neg\xi(x, \hat{r})$ implies that the bounding constraint on some variable $v$ is violated; i.e.

$$\neg B_v(x, \hat{r}) \sqsupseteq \hat{r}(v).$$

To "fix" such a violation we can change the current valset of $v$ to

$$B_v(x, \hat{r}) \sqcap \hat{r}(v),$$

which simultaneously refines $\hat{r}(v)$, since

$$\hat{r}(v) \sqsupseteq B_v(x, \hat{r}) \sqcap \hat{r}(v)$$

and reestablishes the constraint on $v$, since

$$B_v(x, \hat{r}) \sqsupseteq B_v(x, \hat{r}) \sqcap \hat{r}(v).$$

Let

$$B(x, \hat{r}) = \{\!| \; u \rightarrow \; B_u(x, \hat{r}) \sqcap \hat{r}(u) \mid u \in \mathit{domain}(\hat{r}) \; |\!\}$$

```
**Spec** *Global-Search-Program (T :: Global-Search)*

**Operations**

    F-initial-propagate  $(x : D \mid I(x))$
        **returns** $(\hat{t} : \hat{R} \mid \hat{t} = max_{\sqsupseteq} \{\hat{s} \mid top(x) \sqsupseteq \hat{s} \ \wedge \ \hat{I}(x,\hat{s}) \ \wedge \ \xi(x,\ \hat{s})\})$

    F-split-and-propagate
        $(x : D,\ \hat{r} : \hat{R},\ c : \hat{C}$
         $\mid I(x) \ \wedge \ \hat{I}(x,\hat{r}) \ \wedge \ Split\text{–}Arg(x,\hat{r},c) \ \wedge \ \xi(x,\hat{r}) \ \wedge \ \hat{r} \neq bot)$
        **returns** $(\hat{t} : \hat{R} \mid \hat{t} = max_{\sqsupseteq} \{\hat{s} \mid \hat{r} \sqsupseteq \hat{s} \ \wedge \ \hat{I}(x,\hat{s})$
                                                    $\wedge \ Split(x,\hat{r},c,\hat{s}) \ \wedge \ \xi(x,\hat{s})\})$

    F-gs $(x : D,\ \hat{r} : \hat{R} \ \mid \ I(x) \ \wedge \ \hat{I}(x,\hat{r}) \ \wedge \ \Phi(x,\hat{r}))$
        **returns** $(z : R \mid O(x,z) \ \wedge \ Satisfies(z,\hat{r}))$
        $=$ **if** $\exists(z)\,(Extract(z,\hat{r}) \ \wedge \ O(x,z))$
            **then** $some(z)\,(Extract(z,\hat{r}) \ \wedge \ O(x,z))$
            **else** $some(z)\,\exists(c : \hat{C},\ \hat{t} : \hat{R})$
                              $(Split\text{–}Arg(x,\hat{r},c)$
                              $\wedge \ \hat{t} = F\text{-}split\text{-}and\text{-}propagate(x,\hat{r},c) \ \wedge \ \hat{t} \neq bot$
                              $\wedge \ z = F\text{-}gs(x,\hat{t}))$

    F $(x : D \mid I(x))$
        **returns** $(z : R \mid O(x,z))$
        $= some(z)\,\exists(\hat{t}) \ (\hat{t} = F\text{-}initial\text{-}propagate(x)$
                          $\wedge \ \hat{t} \neq bot$
                          $\wedge \ z = F\text{-}gs(x,\hat{t}))$

**end spec**
```

Figure 5: Global Search Program Theory

then, define $f$ as:

$$f(x, \hat{r}) = \hat{r} \sqcap B(x, \hat{r})$$

Constraint propagation is treated here as iteration of $f$ until a fixed-point is reached. Efficiency requires that we go farther, since only a sparse subset of the variables in $\hat{r}$ will be updated at each iteration. If we implemented the iteration on a vector processor or SIMD machine, the overall computation could be fast, but wasteful of processors. On a sequential machine, it is advantageous to analyze the constraints in $\xi$ to infer dependence of constraints on variables. That is, if (the valset of) variable $v$ changes, which constraints in $\xi$ could become violated? This dependence analysis can be used to generate special-purpose propagation code as follows.

For each variable $v$, let *affects(v)* be the set of variables whose constraints could be violated by a change in $v$; more formally, let

$$\mathit{affects}(v) \;=\; \{u \mid v \text{ occurs in } B_u\ \}.$$

We can then generate a set of procedures that carry out the propagation/iteration of $f$: For each variable $v$, generate the following propagation procedure:

```
$Propagate_v\ (x : D,\ \hat{r} : \hat{R},\ new\text{–}valset : VALSET$
                 $\mid I(x)\ \wedge\ \hat{I}(x, \hat{r})$
                 $\wedge\ \hat{r}(v) \sqsupset new\text{–}valset$
                 $\wedge\ B_v(x, \hat{r}) \sqsupseteq new\text{–}valset)$
$=$ let $(\hat{s} : \hat{R} = map\text{–}shadow(\hat{r}, v, new\text{–}valset))$
    if $\neg\hat{I}(x, \hat{s})$ then $bot$
    else
    ... for each variable $u$ in $affects(v)$ ...
    ... generate the following code block ...
    if $\hat{s}\ =\ bot$ then $bot$
    else (if $\neg(B_u(x, \hat{s}) \sqsupseteq \hat{s}(u))$
         then $\hat{s}\ \leftarrow\ Propagate_u(x, \hat{s},\ B_u(x, \hat{s}) \sqcap \hat{s}(u)))$;
    ...
    $\hat{s}$
end
```

where $map\text{–}shadow(\hat{r}, v, new\text{–}valset)$ returns the map $\hat{r}$ modified so that $\hat{r}(v) = new\text{–}valset$.

To finish up, if $Split(x, \hat{r}, i, \hat{s})$ has the form

$$\hat{s}(u) = C(x, \hat{r}, i)$$

for some function $C$ that yields a refined valset for variable $u$, then we can satisfy *F–split–and–propagate* as follows:

$$F\text{–}split\text{–}and\text{–}propagate(x, \hat{r}, i) = propagate_u(\hat{r}, C(x, \hat{r}, i)).$$

The change to $u$ induced in the call to $propagate_u$ will in turn trigger changes to other variables, and so on.

We have described the generation of constraint propagation in a relatively simple setting. One of the authors (Westfold) was largely responsible for the development and implementation of this work. The implementation treats a much broader range of problem features than has been described above, including

1. Heterogeneous variables (and semilattice/refinement structure)

2. Multiple constraints on each variable

3. Indexed variables

4. Conditional constraints

5. Dynamic set of variables

6. Ordering of constraints in propagation procedures

There are many ways to implement constraint propagation, this being just one. Our approach is useful when the *affects* relation is relatively sparse, so special control code to follow dependences and fixing violations is efficient. An alternative approach is to reify *affects* via explicit links between variables, forming a constraint network. The synthesis of the propagation control strategy is relatively simple, since we only need to follow dependence links. Disadvantages of this approach include the size of the constraint network and the cost of maintaining it. This is a common approach in the CSP literature.

Our model of constraint propagation generalizes the concepts of cutting planes in the Operations Research literature [23] and the forms of propagation studied in the constraint satisfaction literature (e.g. [17]). Our use of fixed-point iteration for constraint propagation is similar to Paige's work on fixed-point iteration in RAPTS [7]. The main differences are (1) RAPTS expects the user to supply the monotone function as part of the initial specification whereas we derive it from a more abstract statement of the problem; (2) RAPTS instantiates a straightforward iteration scheme and then performs optimizations. Such an approach would be too inefficient for scheduling applications, so we use dependence analysis to generate code that is specific to the constraint system at hand.

*Constraint Propagation for Transportation Scheduling*

For transportation scheduling, each iteration of the *Propagate* operation has the following form, where $est_i$ denotes the earliest-start-time for trip $i$ and $est_i'$ denotes the next value of the earliest-start-time for trip $i$ (analogously, $lst_i$ denotes latest-start-time), and $roundtrip_i$ is the roundtrip time for trip $i$ on resource $r$. For each resource $r$ and the $i^{th}$ trip on $r$,

$$est'_i = max \begin{cases} est_i \\ est_{i-1} + roundtrip_i \\ max\text{--}release\text{--}time(manifest_i) \end{cases}$$

$$lst'_i = min \begin{cases} lst_i \\ lst_{i+1} - roundtrip_i \\ min\text{--}finish\text{--}time(manifest_i) \end{cases}$$

Here $max\text{--}release\text{--}time(manifest_i)$ computes the max over all of the release times of movement requirements in the manifest of trip $i$ and $min\text{--}finish\text{--}time(manifest_i)$ computes the minimum of the finish times of movement requirements in the same manifest. Boundary cases must be handled appropriately.

After adding a new movement record to some trip, the effect of *Propagate* will be to shrink the

$$\langle earliest\text{--}start\text{--}time, latest\text{--}start\text{--}time\rangle$$

window of each trip on the same resource. If the window becomes negative for any trip, then the partial schedule is necessarily infeasible and it can be pruned.

The constraint propagation code generated for $TS$ in Appendix D in [41] is nearly as fast as handwritten propagation code for the same problem (cf. Appendix C in [36]).

### 4.2.4. Constraint Relaxation

Many scheduling problems are overconstrained. Overconstrained problems are typically handled by relaxing the constraints. The usual method, known as Lagrangian Relaxation [23], is to move constraints into the objective function. This entails reformulating the constraint so that it yields a quantitative measure of how well it has been satisfied.

Another approach is to relax the input data just enough that a feasible solution exists. To test this approach, we hand-modified one version of KTS so it relaxes the LAD (Latest Arrival Date) constraint. The relaxation takes place only when there is no feasible solution to the problem data. KTS keeps track of a quantitative measure of each LAD violation (e.g. the difference between the arrival date of a trip and the LAD of a movement requirement in that trip). If there is no feasible reservation for the movement requirement being scheduled, then KTS uses the recorded information to relax its the LAD. The relaxation is such as to minimally delay the arrival of the requirement to its POD.

This technique, which we call *data relaxation*, can be described more generally. Suppose that we specify a certain constraint to be relaxable. Whenver we detect that the input data has no feasible solution, we attempt to relax the input data just enough to allow a feasible solution. Of course, the problem-solving process and data relaxation are interleaved.

| Data Sets (Air only) | # of input TPFDD records (ULNs) | # of individual movements | # of scheduled items after splitting | Solution time | Msec per scheduled item |
|---|---|---|---|---|---|
| CDART | | 296 | 330 | 0.5 sec | 1.5 |
| CSRT01 | 1,600 | 1,261 | 3,557 | 44 sec | 12 |
| 096-KS | 20,400 | 4,644 | 6,183 | 86 sec | 14 |
| 9002T Borneo | 28,900 | 10,623 | 15,119 | 290 sec | 20 |

Figure 6: KTS Scheduling Statistics

At each global search iteration we evaluate this objective function for all candidate solutions. Using these values the algorithm takes a greedy decision of which branch of the global search tree should be split next. The result is a heuristically-guided algorithm that finds good but not necessarily optimal schedules.

It remains an open task to formalize the notion of data relaxation and to develop a tactic for synthesizing relaxation code in the context of global search with constraint propagation.

### 4.2.5. Using KIDS

In developing a new scheduling application, most of the user's time is spent building a theory of the domain. Our scheduling theories have evolved over months of effort into 50-70 pages of text. It currently takes about 90 minutes to transform our most complex scheduling specification (for ITAS) into optimized and compiled CommonLisp code for Sun workstations. Evolution of the scheduler is performed by evolving the domain theory and specification, followed by regeneration of code.

Currently, the global search deisgn tactic in KIDS is used to design an algorithm for $F$ and $F{-}gs$ in Figuregs-scheme. A specialized tactic for generating constraint propagation code for Horn-like constraints is used to generate code for *F-split-and-propagate*. Once the algorithm is designed, then a series of simplification and common-subexpression-elimination transformations are applied. A trace of the KIDS derivation is given in Appendix D in [41]. See [42] for a detailed description of a session with KIDS.

## 4.3. KTS – Strategic Transportation Scheduling

The KTS schedulers synthesized using the KIDS program transformation system are extremely fast and accurate [44, 45]. The chart in Figure 6 lists 4 TPFDD problems, and for each problem (1) the number of TPFDD lines (each requirement line contains up to several hundred fields), (2) the number of individual movement requirements obtained from the TPFDD line (each line can specify several individual movements requirements), (3) the number of movement requirements obtained after splitting (some requirements are too large to fit on a single aircraft or ship so they must be split), (4) the cpu time to generate a complete schedule, and (5) time spent per scheduled movement. Similar results were obtained for sea movements. The largest problem, Borneo NEO, is harder to solve, because of the presence of 29 movement requirements that are inherently unschedulable: their due date comes before their availability date. Such inconsistencies must be expected and handled by a realistic system. KTS simply relaxes the due date the minimal amount necessary to obtain a feasible schedule.

We compared the performance of KTS with several other TPFDD scheduling systems: JFAST, FLOGEN, DITOPS, and PFE. We do not have access to JFAST and FLOGEN, but these are (or were) operational tools at AMC (Airlift Mobility Command, Scott AFB). According to [10] and David Brown (retired military planner consulting with the Planning Initiative), on a typical TPFDD of about 10,000 movement records, JFAST takes several hours and FLOGEN about 36 hours. KTS on a TPFDD of this size will produce a detailed schedule in *one to three minutes.* So KTS seems to be a factor of about 25 times faster than JFAST and over 250 times faster than FLOGEN on medium-scale problems. The currently operational ADANS system reportedly runs at about the same speed as FLOGEN. When comparing schedulers it is also important to compare the transportation models that they support. KTS has a richer model than JFAST (i.e. handles more constraints and problem features), but less rich than ADANS. The ITAS effort described in the next section reflects our efforts to synthesize schedulers that have at least the richness of the ADANS model.

The DITOPS project at CMU also models scheduling as a constraint satisfaction problem. However, DITOPS effectively interprets its problem constraints, whereas the transformational approach can produce highly optimized "compiled" constraint operations. DITOPS emphasizes complex heuristics for guiding the search away from potential bottlenecks. In contrast KTS uses simple depth-first search but emphasizes the use of strong and extremely fast pruning and constraint propagation code. DITOPS requires minutes to solve the CDART data, whereas KTS finds a complete feasible solution in 0.5 seconds.

*Comparison with PFE* (Prototype Feasibility Estimator, built by BBN based on the Transportation Feasibility Estimator system): On the MEDCOM-SITUATION from the CPE (Common Prototype Environment), KTS is about 5 times faster than PFE and produces a SEA schedule with only 14% of the delay of the PFE schedule. KTS also produces a far more accurate estimate of the planes needed to handle the AIR movements, since PFE is only estimating feasibility whereas KTS produces a detailed schedule.

In our Strategic TPFDD scheduler KTS, we explored issues of speed and embedding KTS into an easy-to-use GUI, complete with ability to edit the data model (TPFDD, resource classes and instances, and port models), to schedule, apply various analysis tools, and to dynamically reschedule. KTS is available from Kestrel via ftp to participants in the PI.

## 4.4. ITAS – In-Theater Airlift Scheduler

In 1994 we began to develop a scheduler to support PACAF (Pacific Air Force) at Hickham AFB, Honolulu which is tasked with in-theater scheduling of a fleet of 26 C-130 cargo aircraft in the Pacific region. Several variants of a theater scheduler, called ITAS for In-Theater Airlift Scheduler, have been developed to date, and more are planned. The system runs on laptop computers (Apple Powerbook). The interface to ITAS and integration with a commercial database package have been developed by BBN. Users enter information about movement requirements, available resources, port features, etc. and ITAS automatically generates a schedule, displayed in a gantt-like "rainbow" chart. The schedule can also be printed in the form of ATO's (Air Tasking Orders).

The ITAS schedulers have emphasized flexibility and rich constraint modeling. The version of ITAS installed at PACAF in February 1995 simultaneously schedules the following types of resources:

1. aircraft
2. aircrews and their duty day cycles
3. ground crews for unloading
4. parking space at ports

each of which may have a variety of attendant constraints and problem features.

# 5. Classification Approach to Algorithm Design

In this section we introduce a new knowledge-based approach to algorithm design. We have been developing it in order to support the incremental application of problem-solving methods to scheduling problems. Our techniques enable us to integrate at a deep semantic level problem-solving methods from Computer Science (e.g. divide-and-conquer, global search), Artificial Intelligence (e.g. heuristic search, constraint propagation, neural nets), and Operations Research (e.g. Simplex, integer programming, network algorithms). Furthermore these techniques have applications far wider than algorithm design, since they apply to the incremental application of *any* kind of knowledge formally represented in a hierarchy.

## 5.1. Technical Foundations – Theories

A *theory* (i.e. first-order theory presentation) defines a language and constrains the possible meanings of its symbols by axioms and inference rules. Theories can be used to express many kinds of software-related artifacts, including domain models [48], formal requirements [2, 9, 24, 26], programming languages [5, 14, 18], abstract data types and modules [9, 13, 16], and abstract algorithms [43]. There has been much work on operations for constructing larger theories from smaller theories [2, 6, 27].

A *theory morphism* translates the language of one theory into the language of another theory in a way that preserves theorems. Theory morphisms underlie several aspects of software development, including specification refinement and datatype implementation [4, 24, 27, 50], the binding of parameters in parameterized theories [8, 14], algorithm design [20, 43, 51], and data structure design [29]. There has been work on techniques for composing implementations in a way that reflects the structure of the source specification [2, 27]; however these composition techniques leave open the problem of constructing primitive morphisms.

Theories together with their morphisms define a category.

## 5.2. Refinement Hierarchy and the Ladder Construction

Abstract programming knowledge can be represented by theories. For example, we showed how to represent divide-and-conquer [30], global search (binary search, backtrack, branch-and-bound) [32], and local search (hillclimbing) [20] as theories. The same approach can be applied to data structures [4], architectures [15], and graphical displays (e.g. Gantt charts).

A collection of problem-solving methods can be organized into a refinement hierarchy using theory morphisms as the refinement arrow [43]. See Figure 5.2.. The question emerges of how to access and apply knowledge in such a hierarchy. The answer is illustrated in the "ladder construction" diagram in Figure 8.

The left-hand side of the ladder is a path in the refinement hierarchy of algorithm theories starting at the root (Problem Theory). $Spec_0$ is a given specification theory of a problem. The ladder is constructed a rung at a time from the top down. The initial arrow (theory morphism) from problem theory to $Spec_0$ is trivial. Subsequent rungs are constructed abstractly as follows:

$$\begin{tikzcd}
P_i \arrow[r, "m_i"] \arrow[d, "I_i"'] & S_i \arrow[d] & \\
P_{i+1} \arrow[r, "m'_{i+1}"] \arrow[rrd, "m_{i+1}"'] & P_{i+1} \oplus S_i \arrow[rd, "m''_{i+1}"] & \\
 & & S_{i+1}
\end{tikzcd}$$

where $P_{i+1} \oplus S_i$ is the pushout theory (shared union) and $S_{i+1}$ is an extension of $S_i$ determined by constructing the theory morphism $m''_{i+1}$. The morphism $m_{i+1}$ is determined by

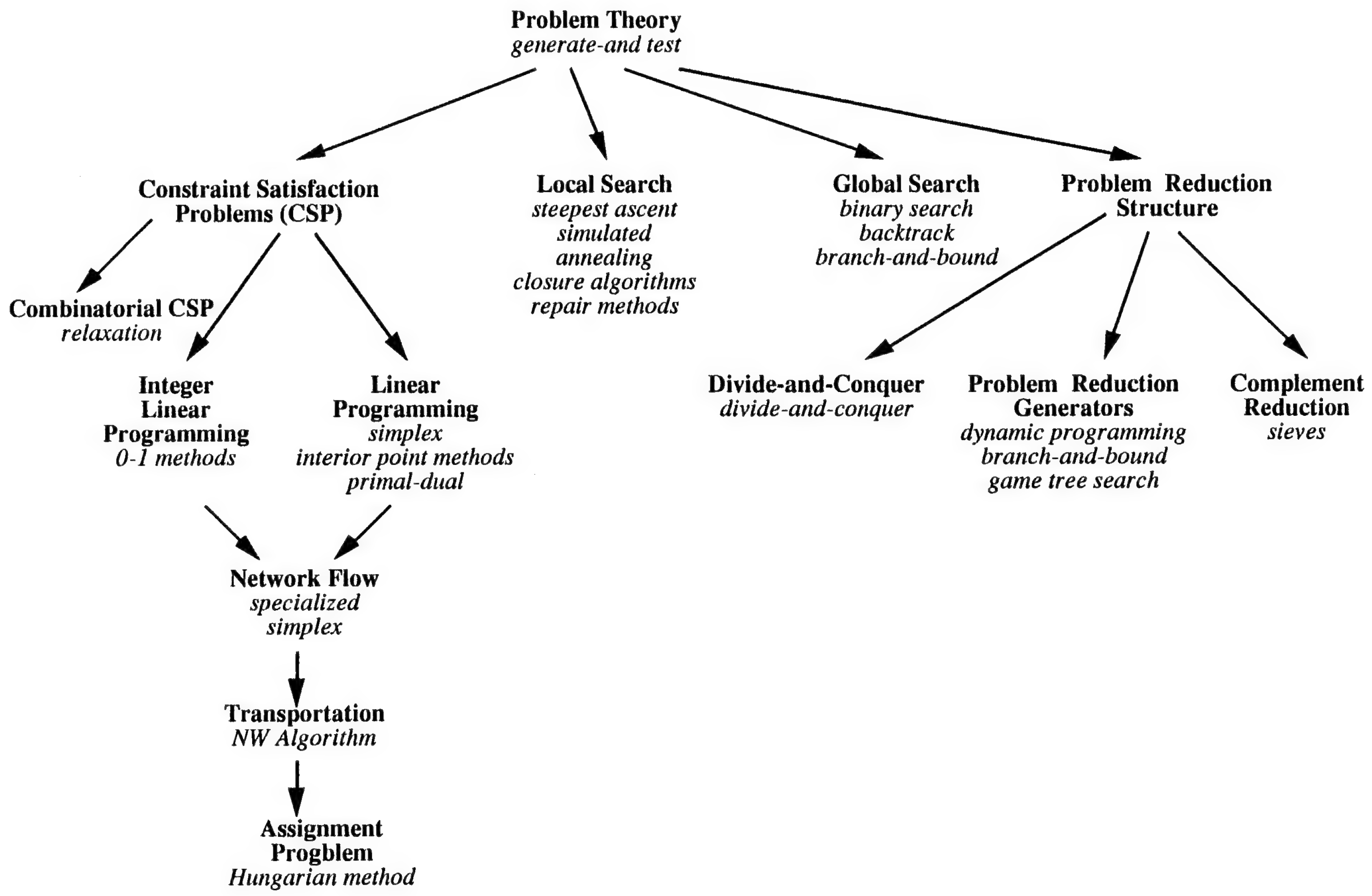

Figure 7: Refinement Hierarchy of Algorithms

$$\begin{tikzcd}
\text{Problem theory} \arrow[r, "m_0"] \arrow[d] & Spec_0 \arrow[d] \\
\text{Constraint Satisfaction} \arrow[r, "m_1"] \arrow[d] & Spec_1 \arrow[d] \\
\text{Mathematical Programming} \arrow[r, "m_2"] \arrow[d] & Spec_2 \arrow[d] \\
\text{Linear Programming} \arrow[r, "m_3"] \arrow[d] & Spec_3 \arrow[d, dashed] \\
\text{Transshipment Problem} \arrow[r, dashed] & ?
\end{tikzcd}$$

Figure 8: Classification Approach to Design

composition.

## 5.3. Constructing Theory Morphisms

Constructing the pushout theory is straightforward. The main issue arising from this ladder construction is how to construct the theory morphism $m''_{i+1}$ from the pushout theory to $S_{i+1}$ (an extension of $S_i$). We formalized four basic methods for constructing theory morphisms last year, by analyzing the algorithm design tactics in KIDS [38]. Two of the techniques are well-known or obvious. However we identified two new general techniques for constructing theory morphisms: unskolemization and connections between theories. Roughly put, *unskolemization* works in the following way. A theory morphism from theory $A$ to theory $B$ is based on a signature morphism which is a map from the symbols of $A$ to the symbols of $B$. A theory morphism is signature morphism in which the axioms of $A$ translate to theorems of $B$. Suppose that during a design process we have somehow managed to construct a partial signature morphism – only some of the symbols of $A$ have a translation as symbols of $B$. The question is how to derive a translation of the remaining symbols of $A$. The unskolemization technique uses the axioms of $A$ and deductive inference to solve for appropriate translations of these symbols. As a simple example, suppose that function symbol $f$ is untranslated and that it is the only untranslated symbol in an axiom $\forall(x)G[f(x)]$ of $A$. We unskolemize $f$ by replacing its occurrence(s) with a fresh existentially quantified variable: $\forall(x)\exists(z)G[z]$. This unskolemized axiom can now be translated and we can attempt to prove it in theory $B$. A proof yields a witness for the existential that is a term that depends on $x$. This term can serve as the translation of $f$ knowing that such a translation preserves the theoremhood of the considered axiom. We may need to verify other axioms involving $f$ to assure the appropriateness of the derived translation.

This technique underlies the problem reduction family of algorithms and tactics in KIDS. For example, in constructing a divide-and-conquer algorithm we need to find translations of decompose, solve, and compose operators. The tactic works by letting the user select a standard decomposition operator from a library (or dually, selecting a standard compose operator) and then using unskolemization on a "soundness axiom" that relates decompose and compose. The unskolemized soundness axiom can then be proved in the given problem theory to yield a specification of the compose (resp. decompose) operator. To be more specific, if we are deriving a divide-and-conquer algorithm for the sorting problem, then we want to construct a theory morphism from divide-and-conquer theory into sorting theory. We might choose a standard decompose operator for input sequences, say split-a-sequence-in-half, and the unskolemization technique leads to a derivation of a specification for the usual merge operation as the translation of compose. The result is a mergesort algorithm. Other choices leads to quicksort, selection sorts, and insertion sorts [30].

Sometimes the axioms of a theory are too complex to allow direct application of unskolemization. This situation arises in the theory of global and local search algorithms. We have discovered and developed recently the concept of *connection between theories* which underlies and generalizes our correct but somewhat ad-hoc solution to this problem in the global and local search design tactics. The general result regarding connections between theories is this:

Suppose that there is a theory $T$ from which we want to construct a theory morphism into a given application domain theory $B$. If there is a (preexisting) theory morphism from $T$ to a library theory $A$ and we can construct a connection from $A$ to $B$, then we immediately have a theory morphism from $T$ to $B$. So connections between theories are a way to adapt a library $T$-theory to a new, but related problem.

The concept of connection between theories relies on several ideas. The sorts of a theory are all interpreted as posets (including booleans) and furthermore the set of sorts itself is a poset (under the subsort partial order). A collection of "polarity rules" are used to express (anti-)monotonicity properties of the functions and predicates of a theory. For example, $size(\{x \mid \neg P\})$ is monotonic in $\{x \mid \neg P\}$ but antimonotonic in $P$; so if $Q \implies P$ then $size(\{x \mid \neg P\}) \leq size(\{x \mid \neg Q\})$. These polarity rules are used to analyze the axioms of a theory and then to set up various connection conditions between the corresponding operators of theories $A$ and $B$ – these conditions directly generalize the conditions of a homomorphism. Furthermore the polarity analysis is used to direct conversion maps between corresponding sorts of $A$ and $B$. Given these conditions and conversion maps it can in general be shown that the axioms of $A$ imply the corresponding axioms of $B$, thus establishing the theory morphism.

We have prototyped this classification approach and have tested it on some simple problems. Steve Westfold built a graphical interface to the refinement hierarchy that allows graphical navigation of it and incremental application. Jim McDonald developed a simple Theory Interpretation Construction Interface that supports the development of views (theory interpretations or morphisms). It shows source and target theory presentations and the current (possibly partial) view between them. Users have several tools to support the completion of a view, including typing in translations for various source theory symbols and using "unskolemization" (one of the four basic methods mentioned above). We demonstrated the use of this system to develop a view from divide-and-conquer theory into a simple problem theory. A more complete implementation of these techniques is underway in the Specware system at Kestrel [49].

## 6. Concluding Remarks

Our work continues to proceed with synergistic interactions between theory and applications. We conclude with one final comment on the scheduling work with respect to the ongoing debate about the concept of *reuse* in software engineering. Our first conception how to develop schedulers was based on reuse of a general-purpose object base manager and the compilation of declarative constraints into object base demons. We also used a Simplex code to check feasibility of start-times in a generated schedule. The results were somewhat disappointing in that for the CDART problem we obtained from CMU, our first code couldn't solve it running overnight, and our second code could only solve most of it using several minutes time. The derived scheduler described in this paper finds a complete feasible solution to the same problem in less than one second.

Since speed is of the essence during the scheduling process and the object base and Simplex algorithm are problem-independent, it seemed wise to exploit our transformational techniques to try to derive codes that are problem-specific and highly efficient. Rather than compile constraints onto an active database, we decided to derive pruning mechanisms and constraint propagation code operating on problem-specific data structures. Rather than use a Simplex algorithm for finding feasible start-times, the constraint propagation code maintains feasible start-times throughout the scheduling process. The advantage of our approach is the ability to expose problem structure and exploit it by transformationally deriving efficient problem-specific code. In this case, the reuse of design knowledge (global search, constraint propagation) as opposed to code components (Simplex, active objectbase) made an enormous difference in performance. We believe that design knowledge, as captured in KIDS and related systems, will be much more reusable than code-level components.

## References

[1] Allen, J. F. Maintaining knowledge about temporal intervals. *Communications of the ACM 26*, 11 (November 1983), 832–843.

[2] Astesiano, E., and Wirsing, M. An introduction to ASL. In *Program Specification and Transformation*, L. Meertens, Ed. North-Holland, Amsterdam, 1987, pp. 343–365.

[3] Biefeld, E., and Cooper, L. Operations mission planner. Tech. Rep. JPL 90-16, Jet Propulsion Laboratory, March 1990.

[4] Blaine, L., and Goldberg, A. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 165–204.

[5] Broy, M., Wirsing, M., and Pepper, P. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems 9*, 1 (January 1987), 54–99.

[6] Burstall, R. M., and Goguen, J. A. Putting theories together to make specifications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (Cambridge, MA, August 22–25, 1977), IJCAI, pp. 1045–1058.

[7] Cai, J., and Paige, R. Program derivation by fixed point computation. *Science of Computer Programming 11* (1989), 197–261.

[8] Ehrig, H., Kreowski, H. J., Thatcher, J., Wagner, E., and Wright, J. Parameter passing in algebraic specification languages. In *Proceedings, Workshop on Program Specification* (Aarhus, Denmark, Aug. 1981), vol. 134, pp. 322–369.

[9] Ehrig, H., and Mahr, B. *Fundamentals of Algebraic Specification, vol. 2: Module Specifications and Constraints.* Springer-Verlag, Berlin, 1990.

[10] et al., J. S. *A Review of Strategic Mobility Models and Analysis.* Rand Corporation, Santa Monica, CA, 1991.

[11] FOX, M. S., SADEH, N., AND BAYKAN, C. Constrained heuristic search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (Detroit, MI, August 20–25, 1989), pp. 309–315.

[12] FOX, M. S., AND SMITH, S. F. ISIS – a knowledge-based system for factory scheduling. *Expert Systems 1*, 1 (July 1984), 25–49.

[13] GOGUEN, J. A., THATCHER, J. W., AND WAGNER, E. G. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, Vol. 4: Data Structuring*, R. Yeh, Ed. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[14] GOGUEN, J. A., AND WINKLER, T. Introducing OBJ3. Tech. Rep. SRI-CSL-88-09, SRI International, Menlo Park, California, 1988.

[15] GRAVES, H. Lockheed environment for automatic programming. Tech. rep., Lockheed Palo Alto Research Center, Palo Alto, CA, 1990.

[16] GUTTAG, J. V., AND HORNING, J. J. The algebraic specification of abstract data types. *Acta Inf. 10* (1978), 27–52.

[17] HENTENRYCK, P. V. *Constraint Satisfaction in Logic Programming.* Massachusetts Institute of Technology, Cambridge, MA, 1989.

[18] HOARE, C. Varieties of programming languages. Tech. rep., Programming Research Group, University of Oxford, Oxford, UK, 1989.

[19] LADKIN, P. Specification of time dependencies and synthesis of concurrent processes. In *9th International Conference on Software Engineering* (Monterey, CA, March 30–April 2, 1987), pp. 106–116. Technical Report KES.U.87.1, Kestrel Institute, March 1987.

[20] LOWRY, M. R. Algorithm synthesis through problem reformulation. In *Proceedings of the 1987 National Conference on Artificial Intelligence* (Seattle, WA, July 13–17, 1987).

[21] LOWRY, M. R. *Algorithm Synthesis Through Problem Reformulation.* PhD thesis, Computer Science Department, Stanford University, 1989.

[22] MINTON, S., JOHNSON, M., PHILIPS, A. B., AND LAIRD, P. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the Eighth National Conferenceon Artificial Intelligence* (1990), pp. 290–295.

[23] NEMHAUSER, G. L., AND WOLSEY, L. A. *Integer and Combinatorial Optimization.* John Wiley & Sons, Inc., New York, 1988.

[24] PARTSCH, H. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag, New York, 1990.

[25] SADEH, N. Look-ahead techniques for micro-opportunistic job shop scheduling. Tech. Rep. CMU-CS-91-102, Carenegie-Mellon University, March 1991.

[26] SANNELLA, D., AND TARLECKI, A. Extended ML: An institution-independent framework for formal program development. In *Category Theory and Computer Programming, LNCS 240* (1985), pp. 364–389.

[27] SANNELLA, D., AND TARLECKI, A. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica 25*, 3 (1988), 233–281.

[28] SELMAN, B., LEVESQUE, H., AND MITCHELL, D. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conferenceon Artificial Intelligence* (1992), pp. 440–446.

[29] SMITH, D. R. Data structure design. in preparation.

[30] SMITH, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27*, 1 (September 1985), 43–96. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).

[31] SMITH, D. R. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming 8*, 3 (June 1987), 213–229.

[32] SMITH, D. R. Structure and design of global search algorithms. Tech. Rep. KES.U.87.12, Kestrel Institute, November 1987.

[33] SMITH, D. R. KIDS: A knowledge-based software development system. In *Automating Software Design*, M. Lowry and R. McCartney, Eds. MIT Press, Menlo Park, 1991, pp. 483–514.

[34] SMITH, D. R. Structure and design of problem reduction generators. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 91–124.

[35] SMITH, D. R. Track assignment in an air traffic control system: A rational reconstruction of system design. In *Proceedings of the Seventh Knowledge-Based Software Engineering Conference* (McLean, VA, September 1992), pp. 60–68.

[36] SMITH, D. R. Transformational approach to scheduling. Tech. Rep. KES.U.92.2, Kestrel Institute, November 1992.

[37] SMITH, D. R. Classification approach to design. Tech. Rep. KES.U.93.4, Kestrel Institute, 1993.

[38] SMITH, D. R. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming 15*, 5-6 (May-June 1993), 571–606.

[39] SMITH, D. R. Derivation of parallel sorting algorithms. In *Parallel Algorithm Derivation and Program Transformation*, R. Paige, J. Reif, and R. Wachter, Eds. Kluwer Academic Publishers, New York, 1993, pp. 55–69.

[40] SMITH, D. R. Toward the synthesis of constraint propagation algorithms. In *Logic Program Synthesis and Transformation (LOPSTR '93)*, Y. DeVille, Ed. Springer-Verlag, 1993, pp. 1–9.

[41] SMITH, D. R. Synthesis of high-performance transportation schedulers. Tech. Rep. KES.U.95.6, Kestrel Institute, March 1995.

[42] SMITH, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.

[43] SMITH, D. R., AND LOWRY, M. R. Algorithm theories and design tactics. *Science of Computer Programming 14*, 2-3 (October 1990), 305–321.

[44] SMITH, D. R., AND PARRA, E. A. Transformational approach to transportation scheduling. In *Proceedings of the Eighth Knowledge-Based Software Engineering Conference* (Chicago, IL, September 1993), pp. 60–68.

[45] SMITH, D. R., AND PARRA, E. A. Transformational approach to transportation scheduling. In *ARPA/RL Knowledge-Based Planning and Scheduling Initiative: Workshop Proceedings* (Tucson, AZ, February 1994), pp. 205–216.

[46] SMITH, D. R., AND WESTFOLD, S. J. Synthesis of constraint algorithms. In *Principles and Practice of Constraint Programming*, V. Saraswat and P. V. Hentenryck, Eds. The MIT Press, Cambridge, MA, 1995.

[47] SMITH, S. F. The OPIS framework for modeling manufacturing systems. Tech. Rep. CMU-RI-TR-89-30, The Robotics Institute, Carenegie-Mellon University, December 1989.

[48] SRINIVAS, Y. V. Algebraic specification for domains. In *Domain Analysis: Acquisition of Reusable Information for Software Construction*, R. Prieto-Diaz and G. Arango, Eds. IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 90–124.

[49] SRINIVAS, Y. V., AND JÜLLIG, R. Specware:$^{\text{tm}}$ formal support for composing software. Tech. Rep. KES.U.94.5, Kestrel Institute, December 1994. To appear in Proceedings of the Conference on Mathematics of Program Construction, Kloster Irsee, Germany, July 1995.

[50] TURSKI, W. M., AND MAIBAUM, T. E. *The Specification of Computer Programs.* Addison-Wesley, Wokingham, England, 1987.

[51] VELOSO, P. A. Problem solving by interpretation of theories. In *Contemporary Mathematics*, vol. 69. American Mathematical Society, Providence, Rhode Island, 1988, pp. 241–250.

[52] ZWEBEN, M., DEALE, M., AND GARGAN, R. Anytime rescheduling. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control* (San Diego, CA, November 5–8, 1990), DARPA, pp. 215–219.